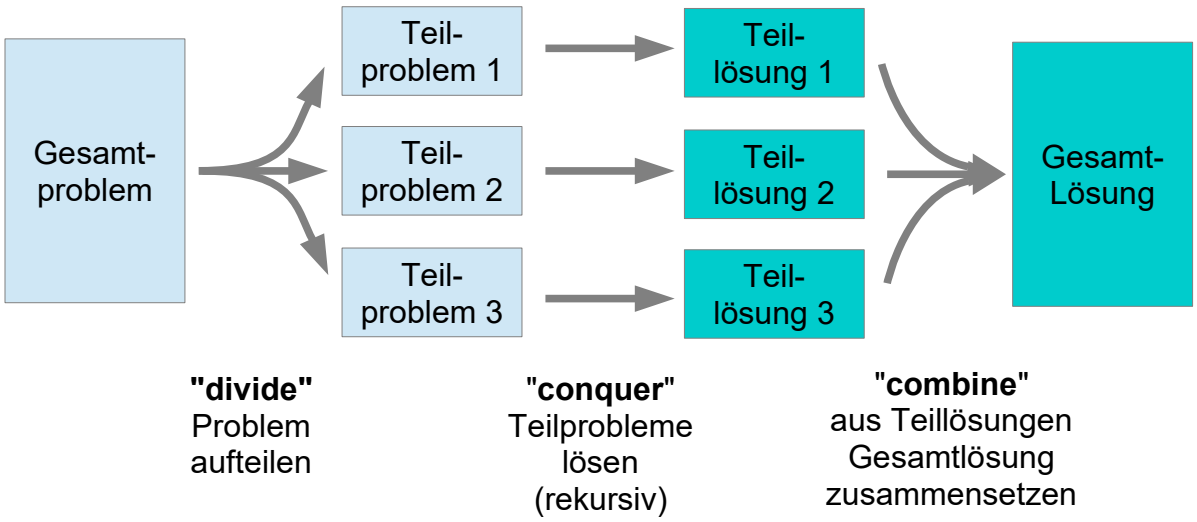
Lektion 9

In der vorigen Lektion haben Sie mit Heapsort ein erstes effizientes Sortierverfahren kennengelernt. In dieser Lektion wird mit Quicksort ein weiterer effizienter Sortieralgorithmus vorgestellt. Gemeinsam mit Mergesort, das in der nächsten Lektion behandelt wird, baut es auf dem gleichen algorithmischen Grundprinzip "Teile und herrsche" auf, das üblicherweise rekursiv umgesetzt wird.

5.3 Rekursives Sortieren nach Prinzip "Teile und herrsche"

Das Algorithmen-Entwurfsprinzip "Teile und herrsche" (*engl*. "divide and conquer", *lat*. "divide et impera") besagt, dass ein schwieriges, nicht direkt lösbares Problem in kleinere gleichartige Teilprobleme zerlegt wird, so dass dann die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems kombiniert werden können. Die Teilprobleme werden üblicherweise selbst wieder in gleicher Weise bearbeitet, so dass sich ein rekursiver Algorithmus ergibt.



Rekursives Sortieren mit Divide-and-conquer-Strategie

Zum Sortieren lässt sich die Divide-and-conquer-Strategie in folgender Weise anwenden:

(1) **Divide**: Besteht die zu sortierende Datenmenge aus mehr als einem Wert, wird sie in zwei Teilmengen aufgeteilt.

(2) **Conquer**: beide Teilmengen rekursiv sortieren.

(3) **Combine**: sortierte Teilfolgen werden zu einer Gesamtlösung vereinigt

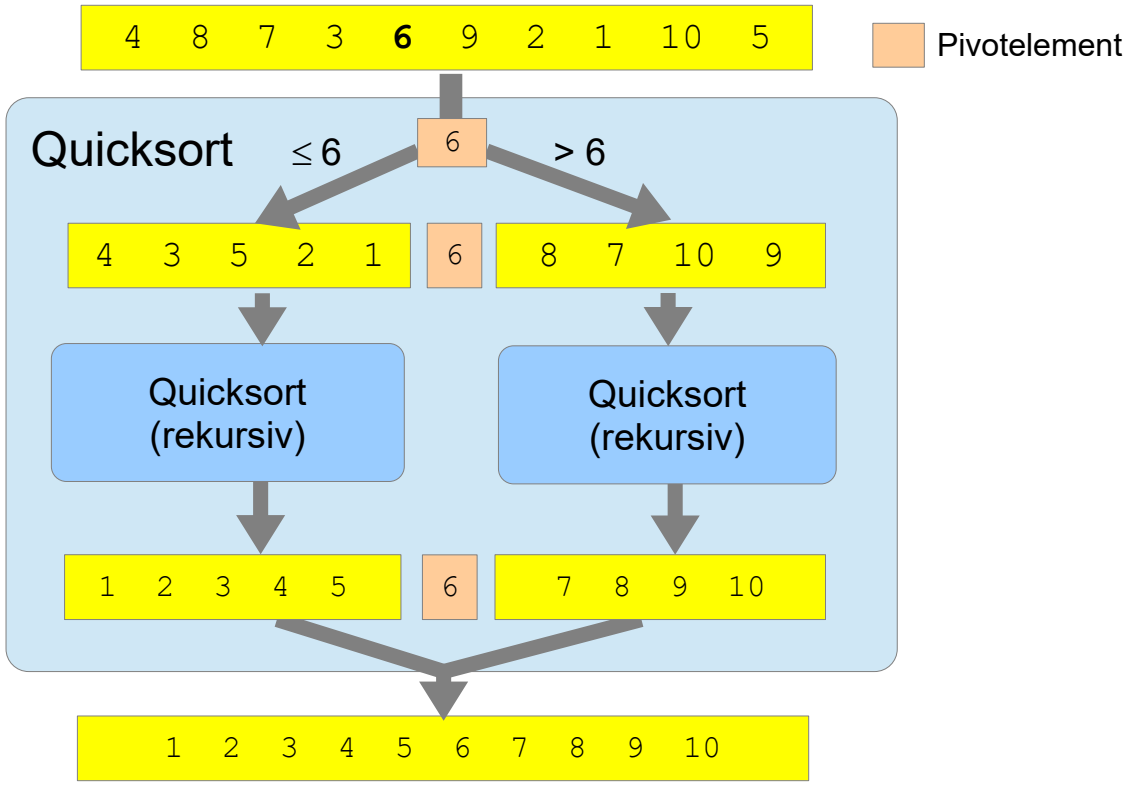
Je nachdem, wie die Teilmengen zerlegt werden und die Teillösungen dann kombiniert werden müssen, ergeben sich verschiedene Sortieralgorithmen:

divide	*combine*	*Sortierverfahren*
Aufteilen in Werte, die größer bzw. kleiner als ein gewähltes Vergleichselement (sog. Pivotelement) sind	Sortierte Teilfolgen aneinander anhängen (bzw. stehen schon nacheinander im Feld), mit Pivotelement dazwischen.	⇒ Quicksort
Aufteilen der zu sortierenden Werte in zwei gleich große Teilmengen (z.B. Arraybereich in der Mitte teilen).	Verschmelzen der sortierten Teilfolgen zu einer sortierten Gesamtfolge durch merge-Operation	⇒ Mergesort

5.4 Quicksort - Sortieren durch Aufteilen

Grundidee von Quicksort

Die Grundidee "Divide and conquer" von Quicksort ist hier an einem Beispiel dargestellt:



Quicksort - Informelle Beschreibung

Sofern der zu sortierende Bereich mehr als ein Element enthält:

(1) Feld in zwei Teile **aufteilen** und Werte dabei umordnen:

- Ein Element des Felds als Vergleichselement (sog. **Pivotelement**) wählen (z.B. Element in der Mitte des zu sortierenden Bereichs)
- Werte, die kleiner als das Vergleichselement sind, kommen in den linken Teil
- Werte, die größer sind, kommen in den rechten Teil
- Die Größe des linken und rechten Teils ergibt sich bei der Aufteilung. Die Teile können unterschiedlich groß werden, ggf. bleibt ein Teil auch ganz leer.

(2) Dann linken und rechten Teil nach gleichem Verfahren sortieren (Rekursion!).

(3) Linker und rechter Teil mit dem Pivot-Element dazwischen bilden zusammen die richtig sortierte Gesamtfolge

Da bei Quicksort die zu sortierende Wertemenge in zwei Teile mit kleineren und größeren Werten aufgeteilt wird, wird Quicksort auch als "Sortieren durch Aufteilen" bezeichnet.

Algorithmus 5.25 - Quicksort/Sortieren durch Aufteilen

```
public static void quicksort(double[] arr) {
      quicksort(arr, 0, arr.length - 1);
}
```

Der Kernpunkt ist folgende rekursive Methode, die den Teilbereich a[links..rechts] sortiert und das oben beschriebene Vorgehen direkt umsetzt:

```
private static void quicksort(double[] a, int links,
                              int rechts){
      if (links < rechts) {
          //mehr als ein Elemente im zu sortierenden Bereich

          //Feldbereich von Index links bis rechts aufteilen
          int pivotPos = aufteilen(a, links, rechts);

          //beide Teile rekursiv sortieren
          quicksort(a, links, pivotPos - 1);
          quicksort(a, pivotPos + 1, rechts);
      }
}
```

5.4.1 Effizientes Aufteilen für Quicksort

Die Herausforderung bei bei Quicksort liegt darin, die Werte effizient in zwei Teilmengen aufzuteilen. Das wurde hier in eine Hilfsmethode `aufteilen` ausgelagert.

Die Methode `aufteilen(a, links, rechts)` wählt das mittlere Element im Bereich `a[links..rechts]` als Pivotwert `pivot` und ordnet die Elemente im Feldbereich so um, dass am Ende

- alle Werte $\leq$ `pivot` links von `ppos` liegen.
- alle Werte > `pivot` rechts von `ppos` liegen.
- der Pivotwert an der Stelle `ppos` zwischen diesen Bereichen liegt.

Die Endposition `ppos` des Pivotwerts wird als Ergebnis zurückgegeben.

Algorithmus 5.26 - Hilfemethode `aufteilen`

```
private static int aufteilen(double[] a, int links,
                                         int rechts){
  // Pivotwert bestimmen
  int mitte = (links + rechts) / 2;
  double pivot = a[mitte];

  // Pivotwert wird vorläufig ganz nach rechts getauscht
  vertausche(a, mitte, rechts);

  int ppos = links;  //am Ende Position für Pivotelement
  for(int i = links; i < rechts; i++) {
      if (a[i] <= pivot) {
          vertausche(a, ppos, i);
          ppos++;
      }
    }             }

  // Pivotwert wird zurückgetauscht an Position ppos, zwischen beide Hälften
  vertausche(a, ppos, rechts);

      //Endposition des Pivotelements zurückgeben
  return ppos;
}
```

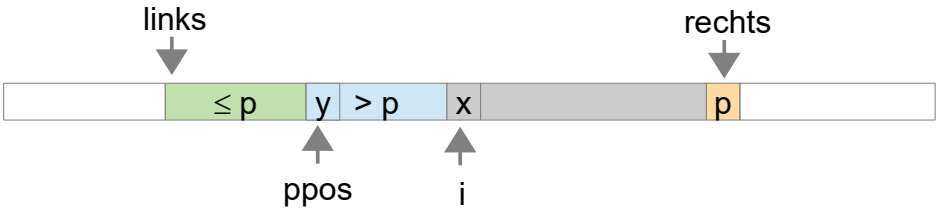
Algorithmus 5.27 - Hilfsmethode `vertausche`

```
static void vertausche(double[] arr, int i, int j) {
```

```
        double tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
}
```

Obwohl diese Methode von der Programmstruktur her mit nur einer Schleife und einer Fallunterscheidung recht einfach ist, ist gar nicht so leicht zu sehen, wie das Aufteilen funktioniert.

Verständlicher wird die Arbeitsweise, wenn man sich folgende Invariante für die for-Schleife klar macht:



(1) Das gewählte Pivotelement p ist am Ende des Bereichs "zwischengeparkt".

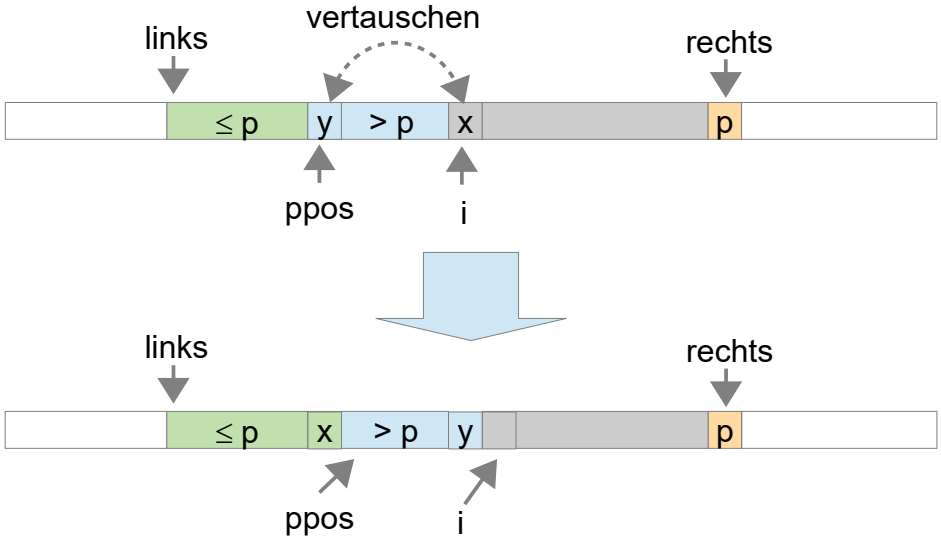
(2) Alle Elemente von Index `links` bis `ppos-1` sind kleiner-gleich dem Pivotwert

(3) Alle Elemente im Indexbereich von `ppos` bis `i`-1 sind größer als der Pivotwert

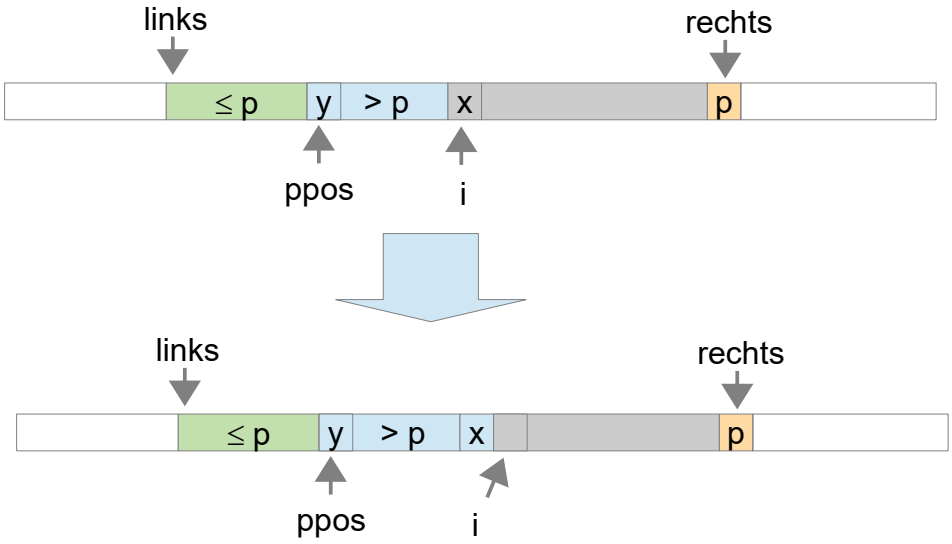
Vor Beginn der Schleife ist diese Invariante offensichtlich erfüllt, da `ppos` = `links`, somit sind die Bereiche für ≤ p und > p noch leer.

Die nachfolgende Betrachtung zeigt, dass die Invariante auch bei jedem Durchlauf erhalten bleibt:

Fall a[i] ≤ pivot: Der Wert x = a[i] wird durch Tausch in die linke Hälfte gebracht. Die rechte Hälfte wird durch den Tausch nach rechts "verschoben":

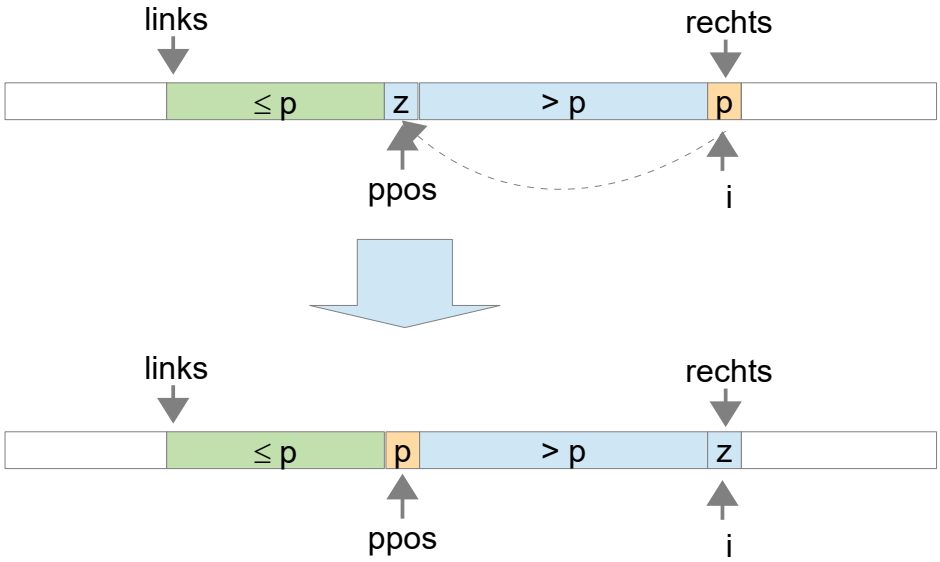


Fall a[i] > pivot: Der Wert x = a[i] grenzt schon an die richtige, rechte Hälfte. Es muss nur i erhöht werden, damit x zur richtigen Hälfte gezählt wird.



Wenn die Schleife beendet wird, gilt also immer noch die Schleifeninvariante (außerdem ist i ≤ rechts auch Invariante, so dass bei Schleifenabbruch i = rechts gilt).

Die Situation nach der Schleife sieht dann wie folgt aus.



Durch das Zurücktauschen des zwischengeparkten Pivotelements landet es zwischen den beiden Hälften. Am Ende der `aufteilen`-Methode sind somit alle Werte in `a[links ..ppos-1]` ≤ `pivot` und alle Werte in `a[ppos+1..rechts]` > `pivot`.

Die dabei verwendete Methode `vertausche` vertauscht in üblicher Weise die Elemente arr[i] und arr[j].

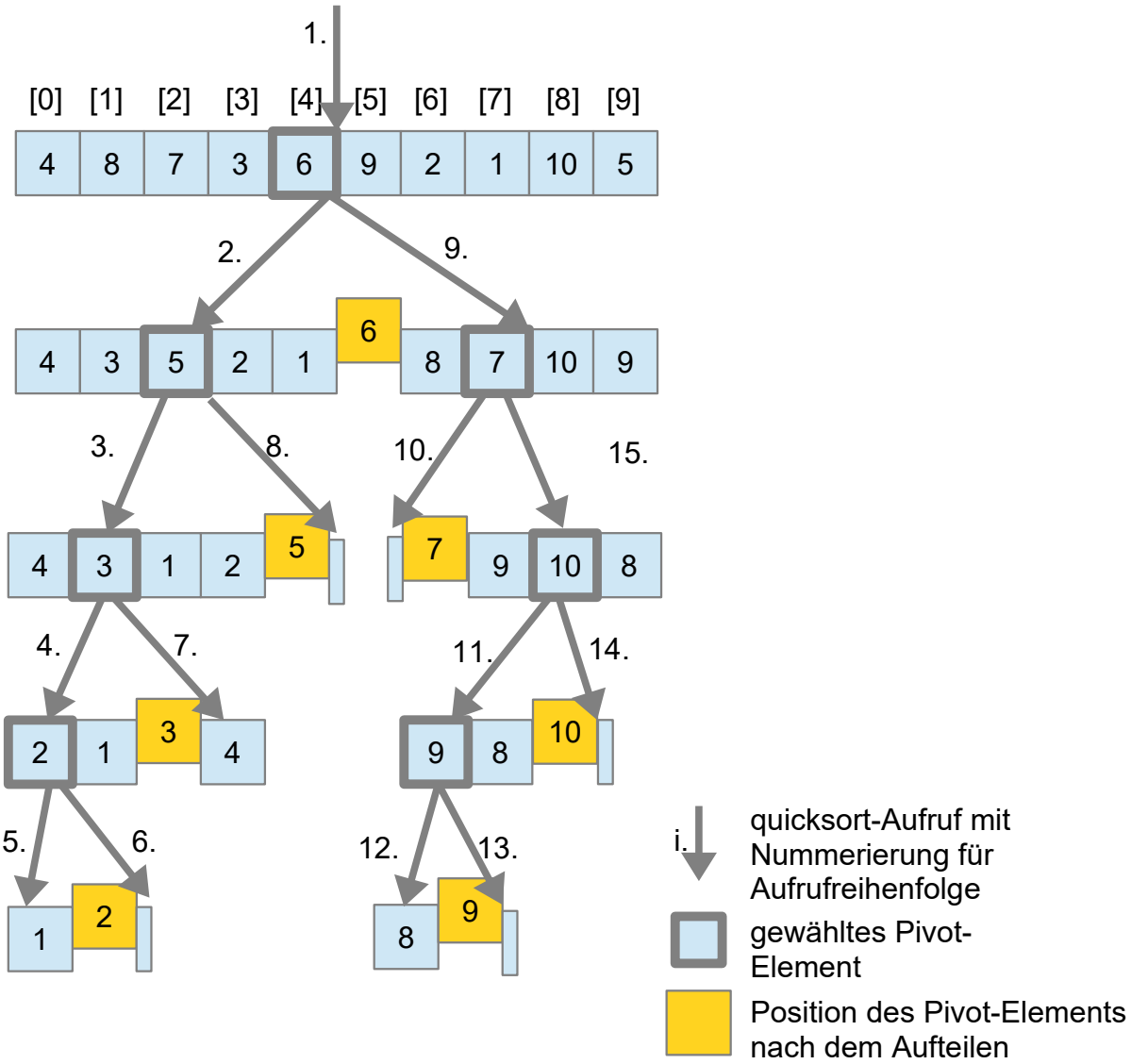
Aufgabe 5.28 - Methode aufteilen()

Vollziehen Sie schrittweise nach, wie die Methode `aufteilen` einen Bereich mit folgenden Werten aufteilen würde:

5, 8, 1, 6, 7, 4, 2

Beispiel 5.29 - Quicksort

Folgende Abbildung zeigt den gesamten Ablauf der rekursiven Aufrufe zum Sortieren eines Arrays.



Aufgabe 5.30 - Quicksort

Vollziehen Sie in analoger Weise nach, wie ein Array mit den Werten

5, 8, 1, 6, 7, 4, 2

mit Quicksort sortiert wird. Das Aufteilen muss dabei nicht genau entsprechend der zuvor angegebenen Methode `aufteilen` erfolgen. Es sollte aber immer das Element in der Mitte des aufzuteilenden Bereichs als Pivot-Wert gewählt werden. Die Reihenfolge der Werte in der linken bzw. rechten Hälfte muss aber nicht genau mit der übereinstimmen, die die Methode `aufteilen()` liefern würde.

Anmerkungen

- Die Urversion von Quicksort wurde 1960 von Sir Charles Antony Richard ("Tony") Hoare erfunden.
- Sie finden in Veröffentlichungen unterschiedliche Implementierungsvarianten für das Aufteilen. Die hier verwendete Implementierung ist eine neuere Version.

C.A.R. Hoare (*1934)
Turing Award 1980

5.4.2 Laufzeitkomplexität von Quicksort

Komplexität der Methode `aufteilen`

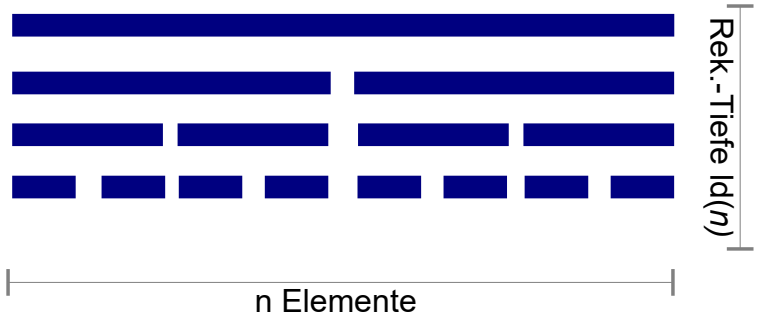
Soll ein Bereich, der n Element umfasst, aufgeteilt werden, muss der Bereich einmal durchlaufen werden und für jeden Wert jeweils ein Vergleich und ggf. eine Vertauschung gemacht werden. Der Aufwand für das Aufteilen eines Bereichs von n Werten ist somit $\Theta(n)$.

Gesamtkomplexität von Quicksort

Beim Aufteilen ist nicht sicher, dass sich zwei ungefähr gleich große Teilbereiche ergeben. Wie Sie gleich sehen werden, hat dies einen wesentlichen Einfluss auf das Gesamtverhalten.

- **Bester Fall**: Es entstehen beim Aufteilen immer zwei weitgehend gleich große Teile, d.h. der zu sortierender Bereich wird in jedem Schritt halbiert.

  Die Bereiche, die aufzuteilen sind, sind hier durch die blauen Bänder visualisiert:



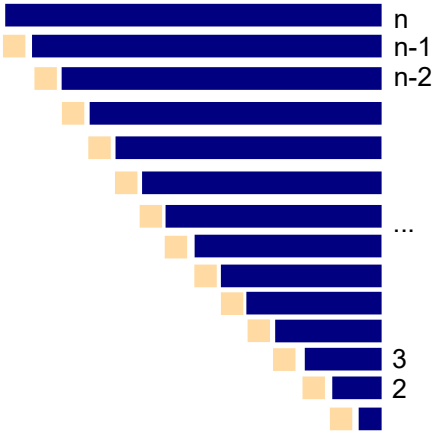
  Da jeweils die Größe der Bereiche halbiert wird, gibt es ca. ld(n) = $\Theta(\log n)$ "Schichten", bis man bei einzelnen Werten landet und die Rekursion abbricht.

  Pro "Schicht", d.h. über alle Aufrufe hinweg für eine Rekursionstiefe, sind jeweils insgesamt ca. n Elemente aufzuteilen (verteilt auf mehrere Aufrufe).
  $\Rightarrow$ Aufwand $\Theta(n)$ pro Schicht

  Somit ergibt sich als Gesamtaufwand für Quicksort im besten Fall:

$$T_{best}(n) = \Theta(\log n)\cdot\Theta(n) = \mathbf{\Theta(n \log n)}$$

- **Schlechtester Fall:** Beim Aufteilen von k Elementen entsteht jeweils ein leerer Teil, der andere Teil enthält dann k-1 Werte. Das ist hier visualisiert.



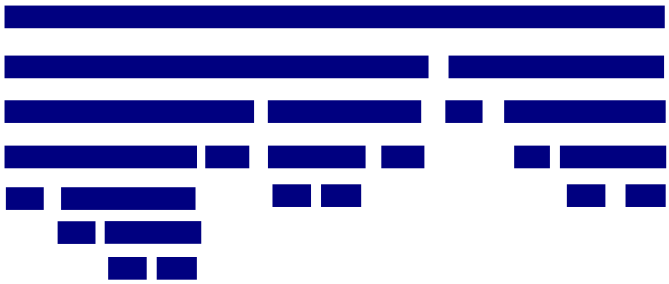
  Zum Aufteilen sind somit insgesamt n + (n-1) + (n-2) … + 2 Schleifendurchläufe und Vergleiche erforderlich. Somit ergibt sich (arithmetische Reihe, Gaußsche Summenformel) insgesamt ein Aufwand in Größenordnung $\Theta(n^2)$ für das Aufteilen über alle Schichten hinweg. Die restlichen Zeitanteile für die rekursiven Aufrufe können im Vergleich dazu vernachlässigt werden. Somit ergibt sich für den

schlechtesten Fall:

$$T_{worst}(n) = \Theta(n^2)$$

Im schlechtesten Fall ist Quicksort also alles andere als "quick", sondern genauso langsam wie die ineffizienten einfachen Sortierverfahren.

- **Mittlerer Fall**: Im mittleren Fall wird das Aufteilen manchmal günstig ablaufen und weitgehend gleich große Hälften erzeugen, manchmal auch weniger günstig und ungleich große Teilbereiche liefern, so wie hier skizziert:



Es lässt sich zeigen, dass auch im mittleren Fall die maximale Rekursionstiefe noch $\Theta(\log n)$ ist. Da pro Rekursionstiefe ("Schicht") der Aufwand für das Aufteilen maximal $\Theta(n)$ ist, ergibt sich für den mittleren Fall die gleiche Größenordnung wie für den besten Fall:

$$T_{avg}(n) = \Theta(\log n)\cdot\Theta(n) = \Theta(n \log n)$$

Anmerkungen

- Quicksort ist ein in-place-Verfahren, d.h. es wird beim Sortieren kein Zusatzspeicher für die zu sortierenden Daten benötigt (bis auf lokale Hilfsvariablen und den Aufrufstack)
- Im Prinzip könnte jedes beliebige Element als Pivot-Element gewählt werden. Das Element in der Mitte zu nehmen hat den Vorteil, dass sich bei schon weitgehend sortierten Daten (was in der Praxis häufiger vorkommt) dann eine günstige Aufteilung in zwei weitgehend gleich große Hälften ergibt.
- Quicksort ist in der Praxis (meist) sehr schnell, da die Anzahl der Datenbewegungen gering ist.
- Bei der hier vorgestellt Version von Quicksort ergibt sich der schlechteste Fall u.a. auch dann, wenn das Array lauter gleiche Werte enthält. Es gibt aber

verbesserte Quicksort-Varianten, die mit Folgen gleicher Werte problemlos zurecht kommen.

- Der "worst case" ist in der Praxis zwar unwahrscheinlich, kann aber auftreten.

Einige Varianten von Quicksort

Einige Erweiterungen von Quicksort, insbesondere auch, um die Wahrscheinlichkeit für den ineffizienten schlechtesten Fall zu minimieren, sind hier kurz skizziert:

(1) **Randomized quicksort**: Das Pivot-Element wird nicht immer aus der Mitte genommen, sondern die Position wird zufällig ausgewählt.
Vorteil: Die Wahrscheinlichkeit für den worst-case-Fall wird dadurch stark verringert.
Nachteil: Es kostet merklich Zeit, jeweils eine Zufallsposition zu berechnen.

(2) Es werden jeweils drei Werte betrachtet, um das Pivotelement zu bestimmen, z.B. das erste, mittlere und letzte Element des Bereichs, und dann der Median der drei Werte (dar größenmäßig mittlere) genommen.
Vorteil: Die Wahrscheinlichkeit, beim Aufteilen jeweils günstige Pivotelemente zu bekommen, ist größer, die Wahrscheinlichkeit für den worst-case-Fall wird somit verringert
Nachteil: Den Median von drei Werten zu bestimmen erfordert zwar nur zwei Vergleiche, aber auch das verlangsamt etwas die Ausführungszeit.

(3) **Dual-Pivot-Quicksort**: Zwei Pivot-Elemente p_1 und p_2 werden gewählt und es wird in drei Teile $[\, x < p_1 \mid p_1 \leq x \leq p_2 \mid x > p_2]$ aufgeteilt.
Vorteil: reduziert die Anzahl erforderlicher Vertauschungen. Es kann dabei auch erkannt werden, wenn Bereiche mit gleichen Werten entstehen, die nicht weiter sortiert werden müssen.

(4) Der zusätzliche Aufwand bei den oben genannten Quicksort-Erweiterungen macht sich insbesondere beim Aufteilen der vielen kleinen Teilbereiche bemerkbar, die man bei größerer Rekursionstiefe hat. Eine mögliche Optimierung besteht darin, ab einer bestimmten Größe des Bereichs (z.B. 5 - 10 Elemente) auf ein einfaches Sortierverfahren umzuschalten, z.B. auf Insertionsort. Quicksort bricht einfach ab, wenn der Sortierbereich entsprechend klein geworden ist und lässt die Bereiche zunächst unsortiert. Am Ende wird ein einziger Insertionsort-Durchlauf für das gesamte Feld durchführen. Da Werte maximal um 5 - 10 Positionen falsch liegen können, hat der Insertionsort-Durchlauf am Ende die Komplexität $\Theta(n)$, so dass sich das nicht nachteilig auf die Laufzeitkomplexität insgesamt auswirkt.

Fazit zu Lektion 9

Das sollten Sie in dieser Lektion gelernt haben

- Was ist das "Divide-and-conquer"-Prinzip beim Entwurf von Algorithmen?
- Wie kann das Prinzip "Divide-and-conquer" auf das Sortieren angewendet werden?
- Wie funktionieren Quicksort?
- Wie wird bei Quicksort das effiziente Aufteilen von zu sortierenden Daten in zwei Hälften bewerkstelligt?
- Welche Laufzeitkomplexität hat Quicksort?